

Reducing complexity of autonomous control agents for verifiability

Paolo Izzo¹, Hongyang Qu² and Sandor M. Veres³

Abstract—The AgentSpeak type of languages are considered for decision making in autonomous control systems. To reduce the complexity and increase the verifiability of decision making, a limited instruction set agent (LISA) is introduced. The new decision method is structurally simpler than its predecessors and easily lends itself to both design time and runtime verification methods. The process of converting a control agent in LISA into a model in a probabilistic model checker is described. Due to the practical complexity of design time verification the feasibility of runtime probabilistic verification is investigated and illustrated in the LISA agent programming system for verifying symbolic plans of the agent using a probabilistic model checker.

I. INTRODUCTION

In control sciences the concept of autonomous control has emerged as an upgrade of feedback control, where the controller also decides what and how to control in terms of objects, parameters and performance [1], [2].

Attempts towards autonomous decision-making software were initially made through Object Oriented Programming (OOP). A first example of OOP, designed with decision-making in mind, is probably given by Ng and Luk in [3]. A more recent example of research in this direction can be found in [4], where Ridao presents a layered OOP control architecture, with deliberative, control execution and reactive layers. For autonomous control, the OOP framework is usually linked to Hybrid Systems (HSs) modelling software. A few examples applications can be found in [5]–[7].

As a next stage of autonomous decision making, formal description of autonomous agents can be found in [8]–[10]. Most architectures are structured in a layered way, as described in references [11]–[13], namely a *Deliberator layer*, a *Sequencer layer* and a *Controller layer*, with different levels of interaction between the layers according to the structure of the system. An exception against this trend that is worth mentioning is the CLARAty architecture [14], [15] developed by NASA where the Deliberator layer and the Sequencer layer are merged together in continuous replanning schemes.

To beat the complexity of decision making which arose, one of the earliest and best known *behaviour-based architectures* was Brook’s *subsumption architecture* [16]. An interesting recent application of this kind of architecture is the MOOS-IvP project [17], [18], which is mostly implemented for Unmanned Underwater Vehicles (UUVs) and Unmanned

Surface Vehicles (USVs). Mission Oriented Operating Suite (MOOS) [19] is an autonomy *middleware*, structured in a star-like fashion. Every application only interfaces with a central database, and the inter-process communication between applications happens with a Publish/Subscribe policy. Interval Programming (IvP) is a MOOS application that implements a behaviour-based decision-making engine [20]. At the end of every reasoning cycle, the engine solves a multi-objective optimisation problem over all the objective functions generated by the behaviours.

Another popular “anthropomorphic” approach to the implementation of autonomous agents is the Belief-Desire-Intention (BDI) architecture which are implemented in programming [8], [21]. BDI agent architectures are characterised by three large abstract sets: *Beliefs*, *Desires* and *Intentions*. The beliefs set represents the information the agent has about the world, the desires set represents something that the agent *might* want to accomplish and the intentions set represents the set of options that the agent is committed to work towards. The most known implementations of the BDI architecture are the Procedural Reasoning System (PRS) [22], [23] and *AgentSpeak* [24]. *AgentSpeak*, and in particular *Jason* [21], [25] and *Jade* [26], [27], fully embrace the philosophy of Agent Oriented Programming (AOP) [28], offering a Java based interpreter that can be customised according to the designer needs.

In this paper we aim to analyse the Jason reasoning cycle to outline its design disadvantages for verification complexity and propose a new architecture called Limited Instruction Set Agent (LISA), which is based on previous expansions of *AgentSpeak* such as *Jason* and *Jade*, while relying more on external planning processes, abstractions from planning and optimisation for decision making called by the agent. We also propose to use model checking techniques at two levels: design-time and at run-time. For the design time model checking we prove that LISA is implementable as a Discrete-Time Markov Chain (DTMC) and that probabilistic model checking is applicable. Given the lack of definition of the plan selection function in *Jason*, our idea is to improve the architecture with a run-time probabilistic model checking by predicting the outcome of applicable plans and actions.

The remaining sections of the paper provide an abstract model for the reasoning cycle in *Jason* and *LISA*, abstractions to DTMC, principles of applicable plan selection, how to use model checking runtime and conclusions complete the paper.

*This work was supported by Thales UK and The University of Sheffield

¹Paolo Izzo is a PhD student at ACSE, The University of Sheffield pizzol@sheffield.ac.uk

²Dr Hongyang Qu is Senior Research Fellow at ACSE, The University of Sheffield, h.qu@sheffield.ac.uk

³Sandor M. Veres is a Professor of Autonomous Systems at ACSE, The University of Sheffield, s.veres@sheffield.ac.uk

II. THE AGENT REASONING CYCLE

By analogy to previous definitions [8], [9], [29] of AgentSpeak-like architectures, we define our agents as a tuple:

$$\mathcal{R} = \{\mathcal{F}, B, L, \Pi, A\} \quad (1)$$

where:

- $\mathcal{F} = \{p_1, p_2, \dots, p_{n_p}\}$ is the set of all predicates.
- $B \subset \mathcal{F}$ is the total atomic belief set. The current belief base at time t is defined as $B_t \subset B$. At time t beliefs that are added, deleted or modified are considered *events* and are included in the set $E_t \subset B$, which is called the *Event set*. Events can be either *internal* or *external* depending on whether they are generated from an internal action, in which case are referred to as “mental notes”, or an external input, in which case are called “percepts”.
- $L = \{l_1, l_2, \dots, l_{n_l}\}$ is a set of logic-based implication rules.
- $\Pi = \{\pi_1, \pi_2, \dots, \pi_{n_\pi}\}$ is the set of executable plans or *plans library*. Current applicable plans at time t are part of the subset $\Pi_t \subset \Pi$, this set is also named the *Desire set*. A set $I \subset \Pi$ of intentions is also defined, which contains plans that the agent is committed to execute.
- $A = \{a_1, a_2, \dots, a_{n_a}\} \subset \mathcal{F} \setminus B$ is a set of all available actions. Actions can be either *internal*, when they modify the belief base or generate internal events, or *external*, when they are linked to external functions that operate in the environment.

AgentSpeak like languages, including LISA, can be fully defined and implemented by listing the following characteristics:

- *Initial Beliefs.*
The initial beliefs and goals $B_0 \subset \mathcal{F}$ are a set of literals that are automatically copied into the *belief base* B_t (that is the set of current beliefs) when the agent mind is first run.
- *Initial Actions.*
The initial actions $A_0 \subset A$ are a set of actions that are executed when the agent mind is first run. The actions are generally goals that activate specific plans.
- *Logic rules.*
A set of logic based implication rules L describes *theoretical* reasoning to improve the agent current knowledge about the world.
- *Executable plans.*
A set of *executable plans* or *plan library* Π . Each plan π_j is described in the form:

$$p_j : c_j \leftarrow a_1, a_2, \dots, a_{n_j} \quad (2)$$

where $p_j \in B$ is a *triggering predicate*, which allows the plan to be retrieved from the plan library whenever it comes true, $c_j \in B$ is called the *context*, which allows the agent to check the state of the world, described by the current belief set B_t , before applying a particular plan, and $a_1, a_2, \dots, a_{n_j} \in A$ is a list of actions.

For comparison we review the semantics of the reasoning cycle in Jason (see Fig. 1 and [21], [25]) and similar languages and then we will introduce and build on the simplifications by LISA in the next section. The reasoning cycle of Jason can be summarised by four main steps:

1) *Belief base update.*

The agent updates the belief base by retrieving information about the world through perception and communication. The job is done by two functions called *Belief Update Function (BUF)* and *Belief Review Function (BRF)*. The BUF takes care of adding and removing beliefs from the belief base; the BRF updates the set of current events E_t by looking at the changes in the belief base.

2) *Trigger Event Selection.*

For every reasoning cycle, only a single event can be dealt with. For this reason a function called *Event Selection Function* selects one of the events from the current event set E_t :

$$S_E : \wp(B) \rightarrow E_t \quad (3)$$

where $\wp(\cdot)$ is the so called *power operator* and represents the set of all possible subset of a particular set. We will call the current selected event $S_E(E_t) = e_t$.

3) *Plan Selection.*

Once the triggering event to be dealt with in the current reasoning cycle is selected, the agent retrieves from the plan library a set of all executable plans that feature e_t (the current event) as triggering event; then these plans are checked for compatible context. All the plan that meet the triggering event and the context are selected as part of an *Applicable Plans* set Π_t , also called *Desire set*. From the set of applicable plans a function called *Applicable Plan Selection Function* S_O (“O” stands for Option) selects the plan that will actually be used to pursue the selected goal or deal with the selected event:

$$S_O : \wp(\Pi) \rightarrow \Pi \quad (4)$$

We will call the current selected plan $S_O(\Pi_t) = \pi_t$.

4) *Intention Selection.*

Once a plan has been selected, the agent is committed to execute and complete it, in order to achieve its goals. A plan that is selected for execution is called *intention* and it is copied in the *Intentions Set* I . At every reasoning cycle the agent is only able to execute a single action; this means that the intention base is likely to contain multiple plans at any given time. A function called *intention selection function* X_I selects the plan to be executed in the current reasoning cycle:

$$X_I : \wp(\Pi) \rightarrow \Pi \quad (5)$$

The selected intention $X_I(I) = \pi_t^{(i)}$ is then taken on for execution.

5) *Action Execution.*

The agent takes the next action from the plan and call

We now present the LISA agent reasoning cycle, highlighting the differences with the Jason reasoning cycle presented in Section II. The simplified reasoning cycle is illustrated in Fig. 2. Subscript t indicates the indexing of the reasoning cycle.

1) *LISA belief base update.*

At the beginning of every reasoning cycle, the BUF checks for all the input coming from perception, action feedback and communications and updates the current belief set B_t . The BUF function also removes perception and action feedback predicates if they are not received persistently. For instance indication of a task completed may be a feedback sent only once while sensor based detection of dynamical instability can be a persistent feedback. The BRF generates a set of events $E_t = B_t \setminus B_{t-1}$ at every reasoning cycle. The LISA agent does not select a single trigger for intention but executes all plans with true context plans in a multi-threaded way.

2) *Plan selection.*

Next the agent looks at the current event set E_t and retrieves all the plans from the plan library Π that are triggered by these events, it checks that the context meets the current beliefs, and copies the plans to the Desire set $\Pi_t \subset \Pi$.

3) *Action Execution.*

The agents retrieves the next action to be executed from each plan and calls for externals or internal functions to execute the action. Similarly to Jason, a plan is suspended while waiting for a completion feedback from an action.

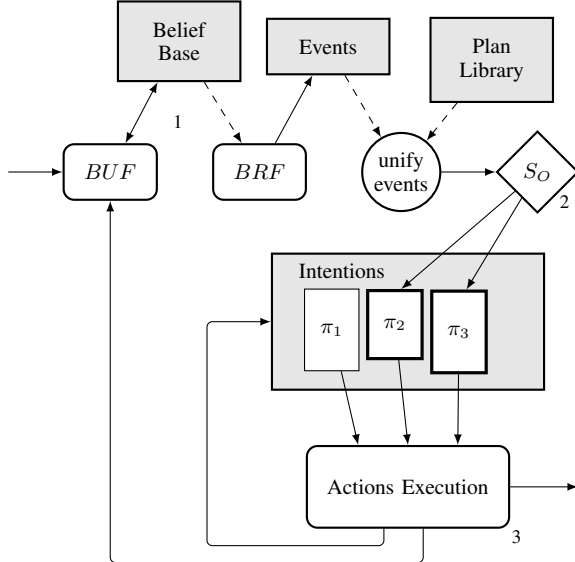


Fig. 2: Reasoning cycle of LISA: the plans run in a multi-threaded way, avoiding the need of S_E and S_I

The LISA agent reasoning cycle reduces complexity compared to previous implementations and the agent decision making process can be easily modelled as a DTMC, with particular attention to the selection of the executable plans,

as proven in the next section.

IV. LISA ABSTRACTIONS TO DTMC

This section will address the problem of modelling and model checking the decision processes of the LISA architecture as mathematically described in the previous section. Model checking is understood here before the application of the agent in the environment, i.e. at design time. The objective is to verify the functionality of the agent code in the environment, assuming the software has been verified to precisely deliver its designed functionality in LISA. The functionality of LISA is to be verified against a series of queries using probabilistic model checking techniques. In this paper, we will be using PRISM [30], [31], a popular probabilistic model checker, to perform the verification. Detailed study of the list of verification queries in PRISM goes beyond the scope of this paper. Here our objective is to consider the specification itself and show that verification can in principle be carried out.

We will assume here that the LISA agent to be verified will function in a physical environment which it needs to perceive and model using sensing instruments such as IMUs, Lidars, Sonars, cameras and so on. While it plans and takes actions, the agent will not always completely succeed with its intentions. It is assumed however that any action process is able to assess its own outcome, i.e. success, partial success or failure, and that it is able to feed this information back to the belief base of the agent. The action feedbacks can potentially include direct information on the cause of problems in case of failure or partial failure, however the level of detail will depend on the amount of the agent programmers' efforts to identify what is necessary to achieve a useful intelligent machine behaviour. This way the decision making of the agent can be gradually upgraded by more and more detailed analysis of its own work. For our verification theory the level of depth in the action-feedback is not relevant but its existence is.

The agent and the environment is considered as a single system to be verified. In fact there will be three subsystems to be modelled for verification purposes: (1) the robotic agent (2) the physical environment (3) the human operator (see Fig. 3). In a broader approach to verification, when we verify

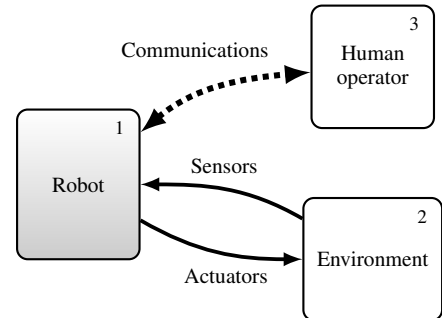


Fig. 3: Subsystems of the full model. The human operator can send mission objectives to the Robot, and receive reasoning feedback. The robot interacts with the environment with an array of sensors and actuators.

the agent for a broader class of environmental possibilities, the action feedbacks are events with random characteristics which do not reflect the internal structure of the environment.

Theorem 1. Assuming an independent probability distribution of all action feedbacks and sensory events from the environment, the complete decision making of a LISA can be modelled by a DTMC.

Proof. (Outline) As during the execution of any action(s) in some plan(s) is(are) represented by specific predicates in B , the power set $\wp(B)$ (the set of subsets of B) forms a state space for the agent as any $s \in \wp(B)$ provides complete information for the decision making of the agent. This holds true regardless of the fact that the memory of the agent (in terms of its temporal world model) influences its decisions, as those decisions are modelled by probabilistic predicate feedback from processes of the agent analysing its external world. The state of the DTMC is initialised by its initial set of beliefs, goals and probabilities of feedback from its initial actions. Transition to a new state of the agent only happens at the end of each reasoning cycle. The new set of predicates for the new state is developed in two steps: (1) by deterministic application of reasoning rules, triggering or closure of new plans and elimination of predicates due to the semantics of LISA's execution (2) probabilistic appearance of perception predicates and action feedback predicates in B . The probability distribution of new sensory and action feedback predicates is well defined in any LISA program and hence modelling as a DTMC can be completed. \square

The fact that we do not need a MDP (Markov Decision Process) to model the LISA agent and that a DTMC suffices, signifies the importance of defining limited instruction set agent for verification purposes to reduce complexity. Nevertheless, a lot can be done to further reduce complexity of design-time verification through programming style in LISA.

Another approach to environmental modelling is to abstract away the simulation of the environment so that perception events and actions feedbacks are modelled by probabilistic dependency in terms of DTMCs. Similarly, the behaviour of human operators can also be formulated in terms of a DTMC. The next formal result covers this case.

Theorem 2. Assuming that both human communication and the physical environment are possible to model by DTMCs, which are possibly inter-dependent by conditional probabilities, the complete decision making of a LISA in its environment can be modelled by a DTMC.

Proof. (Outline) The alteration relative to the proof of Theorem 1 is that the probabilistic appearance of perception and action feedback predicates is this time determined by the DTMCs of human communication and interaction with the agent and predicates of action feedback from the physical environment. Due to the DTMCs for both human responses as well as environmental responses well defined, all the state transition probabilities can be calculated by basic rules of conditional probabilities and the proof is hence com-

pleted. \square

It directly follows from Theorem 1 and 2 that the decision making system of a LISA agent can be verified in PRISM at design time. The following sections will examine whether executable plan set selection can be done in terms of *run-time model checking*, which can also be termed as *runtime verification of executable plan selection* by the agent.

V. PRINCIPLES OF APPLICABLE PLAN SELECTION

In this section we discuss the general definition of a more advanced method for the Applicable Plan Selection Function S_O .

In order to improve the agent's understanding of the world in association with its own internal state, a finite set of so called "operational states" is defined:

$$X = \{x_1, x_2, \dots, x_{n_x}\} \subset \mathcal{F} \quad (6)$$

Operational states are beliefs, predicates of \mathcal{F} , that represent high level states of the agent, for example "Waiting for instructions" or "Exploring area of interest". Operational states are not mutually exclusive, more than one can be active at any given time. We will call $X_t \subset X$ the set of active operational states at time t .

The agent has to take care of updating its own operational state by applying a set of logic implication rules. These rules are based on statements called "pre-conditions". Pre-conditions include beliefs generated by the percept process as well as mental notes created during plans executions.

The associations between plans, that are effectively sequences of actions that the agent can take and their possible outcomes, can mostly be made in advance, especially using advanced simulation software. However for advanced systems such as a fully operational autonomous vehicles, the amount of *possible outcomes* can easily become too high to be accounted for at runtime. In order to limit the number of associations, we apply the following classes of constraints:

- *Temporal order of actions.*

The agent can be programmed so it has the ability to memorise and later check for past actions. Some actions must be preceded by others, for instance the agent is not allowed to pick up an instrument that it never deployed in the environment.

- *Actions performed in the context of operational modes.*

Operational modes allow to give a general context to the status of the mission and can be used to reduce the number of associations between course of actions and possible outcomes .

- *Environmental dependency.*

An autonomous vehicle is designed to operate autonomously in any condition. External events shape the way the system is going to act throughout the mission. For instance if the agent experiences a communication loss with other agents, whom it was supposed to communicate with, the mission objectives may not be achievable any more and the agent may want to do something else instead.

Let us define an “implication function” $f_{\mathcal{I}}$ that associates every plan a finite set of possible outcomes, and the likelihood that each of those event would happen:

$$\mathcal{I} : \Pi \rightarrow \wp(\wp(B) \times [0, 1]) \quad (7)$$

Once all the associations have been made, the agent can generate a graph that represents all the possible course of actions that the agent can go through according to its symbolic plan list for the future, using outcomes (events) associated with actions in the code of the plans. Fig. 4 shows a simple two time steps representation of this concept.

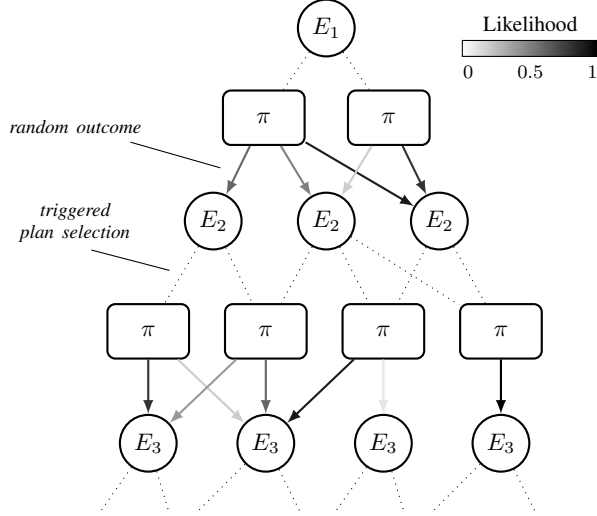


Fig. 4: Example of possible course of plans tree: every plan can generate one or more events set with a certain probability, illustrated by the colour of the arrow. Every set of events in turn can trigger more than one plan.

Let us define a “course of plans” as a sequence of plans $\hat{\Pi} \subset \Pi$, a branch of the tree in Fig. 4, associated with a particular goal represented by a set of events E_t that the agent is committed to pursue, and the likelihood that the course of action leads to the successful achievement of said goal:

$$c_{\pi}^{(j)} = \left\{ \hat{\Pi}^{(j)}, E_t^{(j)}, \lambda^{(j)} \right\} \quad (8)$$

where the index j numbers each branch of the three in Fig. 4. At any moment in time the agent will have some knowledge of the environment in the form of belief predicates, it will know in what operational state it is operating at the moment and it will have knowledge of what has been done in the past, still in the form of predicates. At any particular time a plan can have multiple course of actions associated with it. Using both pre-computed and runtime simulation results, it would be possible to predict the one with the highest likelihood of success to determine whether or not the plan is likely to lead to the achievement of the current goals. To do this “reward value” can be applied to every applicable plan in Π_t . Let us define a “reward function” $f_{\mathcal{R}}$ that uses these principles to assigns a *reward* value to every plan:

$$\mathcal{R}_t : \Pi_t \rightarrow \mathbb{R}^+ \quad (9)$$

This association is clearly time dependant in the sense that the same plan might have a variable likelihood of success value when applied at different points of the mission under different environmental conditions. For this reason, at run time a “reward update function”(which is defined at design time of the agent) needs to be evaluated each time the agent is required to make a choice between different plans, and so different course of actions:

$$\mathcal{RU} : \Pi_{t-1} \times \mathbb{R}^+ \rightarrow \Pi_t \times \mathbb{R}^+ \quad (10)$$

The above runtime schema and method enables the agent to consider it options for the future. In the next section we are going to propose a way to assess its options with a probabilistic model checking technique and finally make the agent to decide about its next action (and likely consecutive actions).

VI. PLAN SELECTION BY RUNTIME MODEL CHECKING

The applicable plan selection function S_O makes a selection amongst plans that are triggered by the same event and that have a context that matches the current belief. The principles proposed in Section V can be modelled with DTMC as shown in Section IV. However in a more complex plan selection mechanism, which may break the applicability of a DTMC model, the S_O block’s plan selection can be replaced by a detailed plan-optimiser or by a combination of a simplified symbolic planner with options at its output and followed by detailed model checker for a time horizon of consequences of actions, which selects the best option. This is not unlike humans select their plans: first they consider a few alternatives and then check each for details. It is

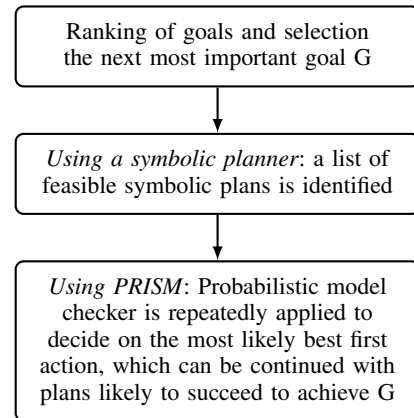


Fig. 5: The principle of selecting the best plan by methods of model checking.

well known that model checking can often be used for plan selection by generating counterexamples for the negation of a temporal logic formula to be verified. On the other hand, planning rarely answers queries about liveness and negative consequences are often hard to formulate as constraints,

especially when indirectly implied, . For these reasons probabilistic model checking, in combination with planning in continuous and discrete space time, is a reasonable choice to check out feasible and most promising next actions and also consequent options for actions depending on how the future outcomes evolve.

Fig. 5 outlines the operation of the runtime model checker for the selection of the next best action. PRISM can generate counterexamples for queries [32], each of which is a trace in the model. The counterexample generation produces the trace with the highest probability first. When we feed PRISM the negation of a query, the first action in the trace, as a part of trace provided counter-example, is the next best action.

In this scheme the agent decides on the next suitable goal using its reasoning rules in the top block of Fig. 5. Next a symbolic planner is applied to generate alternatives of plan and action sequences for the future. This list of symbolic plan/action sequences is then analysed by a probabilistic model checker to find the symbolic plan most likely to be successful.

VII. RUNTIME MODEL CHECKING EXAMPLE IN LISA

Consider an Autonomous Surface Vehicle (ASV) on an exploration mission. The vehicle has to explore two areas, that we will call “Area A” and “Area B”.

The system is trying to carry out two sets of actions:

<i>Explore Area A</i>	<i>Explore Area B</i>
<ul style="list-style-type: none"> Go to Area A Cover full area 	<ul style="list-style-type: none"> Go to Area B Cover full area

Each area is partitioned into a number of blocks: A_1, A_2, \dots, A_{N_A} for Area A and B_1, B_2, \dots, B_{N_B} for Area B. The exploration of the areas can be done in the same number of steps accordingly, one block for each step. The vehicle consumes one unit of fuel to explore one block. During the exploration, the weather can change: with probability p , it becomes bad and with $1 - p$ it becomes good. When the former happens, the vehicle consumes twice as much fuel to explore each block. When the weather in the other area is good, it can move to that area with probability $1 - q$ of a successful passage. When one area is fully explored, the vehicle goes to the other area, assuming there is still enough fuel. Moving between the two areas consumes one unit of fuel, as well as going to each area from the base. At any point during the mission, if the fuel tank becomes almost empty, e.g. when the fuel in the tank is one unit, the vehicle has to go back to the base. Once both areas are explored, the agent will head back to the base.

The symbolic plans are simply extracted from pre-defined plans in LISA by an action of the LISA agent, which generates symbolic plans to satisfy pre and post conditions, including operational state history. The probabilities for the DTMC in PRISM are obtained from the action feedback probabilities in the LISA program. Hence the programming framework also lends itself to automated extraction of PRISM models

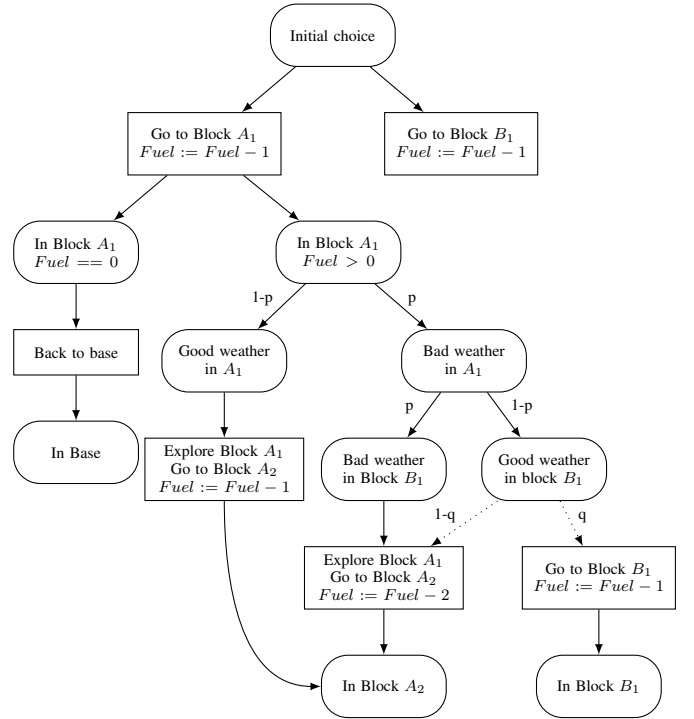


Fig. 6: The partial Abstract DTMC for the example

of runtime verification for decision making. Fig. 6 illustrates a partial graph of the DTMC that models the example for the design time verification. The dotted lines represent the plans that need to be chosen. The reward for each plan can be the probability of successfully fulfilling the mission from the plan, which can be specified as a reachability query. This probability is then computed by PRISM for each state in the DTMC. The reward for the plan “Explore Block A_1 and go to Block A_2 ” is the probability at the state “In Block A_2 ”, while the reward for the other plan is the probability at the state “In Block B_1 ”. The Appendix contains the sample PRISM program for Fig. 6.

VIII. IMPLEMENTATION OF LISA

A suitable framework to implement LISA is MOOS-IvP [17]–[19]. MOOS is a inter-process communications-middleware software that is structured in a star-like fashion. It features a central node called the MOOS Database (MOOSDB) and a set of applications that communicate with each other through the MOOSDB in a publish/subscribe manner. IvP is a MOOS application that optimises behaviour selection in actions of the agent. Behaviours for action execution are modules of the IvP that *compete* over the definition of control values that will be used to act on the environment. For example for control variables such as “direction” and “speed”, behaviours can be “waypoint following”, “collision avoidance” and so on. This method is ideal to reduce complexity of the reasoning as it allows to further abstract action definitions and it significantly reduces the need for nested conditional statements within symbolic plans.

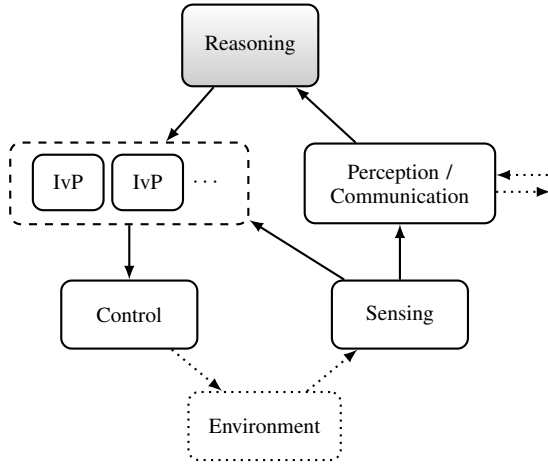


Fig. 7: Implementation of the LISA architecture using IvP planning engines. Every node contains multiple sub-nodes of lower level skills of the LISA agent.

Fig. 7 illustrates the overall structure of the system. Each block is a collection of “skills” of the agent that can be implemented as MOOS processes (nodes). In MOOS the communication between nodes happens through the central database. The *Reasoning* block performs symbolic planning and can activate IvP modules to optimise action execution within plans of the LISA architecture. The *Perception/Communication* block converts filtered sensing data and the communication channels into enriched symbolic information for the agent Reasoning. The IvP modules operate an intermediate feedback loop with some processes within the Sensing and Control blocks through their variables. Processes in the *Control* block interact with physical actuators that operate in the environment. Processes in the *Sensing* block interact with physical sensors and take care of filtering data in order to minimise noise.

Behaviours are internal modules of the IvP application that reproduce a particular action over a set of control variables c_1, c_2, \dots, c_n , and generate at every cycle a piecewise linear function called “IvP function” $f(c_1, c_2, \dots, c_n)$ that maps points of the decision space to values that reflect the degree to which that control array supports the action. Once these functions are produced a multi-objective optimization problem is solved by another internal module called the IvP-solver:

$$\begin{aligned} \arg \max_{c_1, \dots, c_n} \quad & w_1 f_1(c_1, \dots, c_n) + \dots + w_k f_k(c_1, \dots, c_n) \\ \text{s.t.} \quad & f_i \text{ is an IvP piecewise defined function} \\ & w_i \in \mathbb{R}_{\geq 0} \end{aligned} \quad (11)$$

where w_1, w_2, \dots, w_n are called *priority weightings*. In MOOS-IvP the priority weightings are influenced by two main factors:

- 1) With every behaviour is associated a set of binary flags that allow control over the activation time of the behaviour itself. These flags can be conditionally modified by the behaviour itself, which means that the behaviour has partial control over its own state, and

can also be associated with external variables that can be modified by other nodes.

- 2) A hierarchical mode system is defined within IvP that allows to organise the behaviour activation according to declared mission modes. Modes and sub-modes can be declared in line with the designer’s own concept of mission evolution, and conditional statements can be implemented so to switch between modes. Modes can also be associated with external variables that can be modified by other nodes.

The control over the value of the priority weightings allow the Reasoning of LISA an appropriate level of control over the functionality of the behaviours:

- Actions a_1, a_2, \dots, a_{n_a} from the agent set $A \subset \mathcal{F} \setminus B$ can directly activate or deactivate behaviours. At the end of every reasoning cycle (See Fig. 2), LISA takes a set of next actions from the available plans in the Intentions set and issues them for execution. These actions can be external or internal routines. A way to integrate LISA with MOOS-IvP is to create external actions that can modify variables that are linked to the behaviour’s activation flags. For example an action of the type “Go to area A” can be executed in many different ways depending on the state of the environment, on the distance and so on. For this purpose the reasoning can send an activation flag to several behaviours that all operate on the same space (for example “direction” and “speed”) and that compete with each other.
- At any given moment in time, a set X_t of operational states from $X = \{x_1, x_2, \dots, x_{n_x}\}$, defined in Eq. (6), are active in the LISA agent. These operational states are updated and activated according to the mission status. When operational state are set or modified in LISA external actions can be executed to modify variables associated with the hierarchical mode structure of IvP. For example an operational state of the type “Exploring area A” can be associated with a set of behaviours that include for example *area covering*, but at the same time *collision avoidance* and *SLAM*.

IX. CONCLUSIONS

The history of methods for controlling of autonomous vehicles has been briefly reviewed and a new agent architecture called Limited Instruction Set Agent (LISA) has been proposed. The architecture originates from previous AgentSpeak implementations such as Jason and Jade. By reviewing in detail the reasoning cycle and the implementation of Jason, we identified several design features which make verification complex and proposed an alternative architecture that improves verifiability of agent reasoning in physical environments. In particular, we reduced complexity by simplifying the reasoning cycle with the use of multi-threaded processing and proposed the use of model checking techniques on two levels: (1) *Design-time model checking*, we prove that it is possible to abstract the agent to a DTMC and in turn verify the decision making process with existing model checking techniques, (2) *Runtime verification*

of executable plan selection, the agent is able to assess a tree of possible future outcomes and select a plan which is most likely to succeed in reaching the mission goals.

APPENDIX

Example PRISM program of runtime verification in the LISA system.

```

1  dtmc
2
3  const int No = 15;
4  const int Na = 5;
5  const int Nb = 5;
6
7  const double Pa = 0.1; // probability of bad weather in Area A
8  const double Pb = Pa; // probability of bad weather in Area B
9  const double Pi = 0.5; // probability of initial choice between Area A and
10 // Area B
11 const double Ps = 0.6; // probability of switch between Area A and Area B in
12 // case of bad weather
13
14 module robot1
15   al : [0..Na] init 0; // Area A
16   bl : [0..Nb] init 0; // Area B
17   oil : [0..No] init No; // oil level
18   s : [0..3] init 0; // 0: base, 1: Area A, 2: Area B, 3: mission aborted
19
20 // initial choice
21 [] (s = 0) & (al = 0) & (oil > 0)
22 -> Pi: (s'=1) & (oil'=oil-1) + (1-Pi): (s'=2) & (oil'=oil-1);
23
24 // decision in Area A
25 [tick2] (s = 1) & (t = 0) & (al < Na) &
26 (oil > 0) & (w1 = 0) -> (al'=al+1) & (oil'=oil-1);
27 [tick2] (s = 1) & (t = 0) & (al = Na)
28 & (bl < Nb) & (oil > 0) -> (s'=2) & (oil'=oil-1);
29 [tick2] (s = 1) & (t = 0) & (al < Na) & (bl < Nb)
30 & (oil > 1) & (w1 = 1) & (w2 = 1)
31 -> (al'=al+1) & (oil'=oil-2);
32 [tick2] (s = 1) & (t = 0) & (al < Na) & (bl < Nb)
33 & (oil > 1) & (w1 = 1) & (w2 = 0)
34 -> Ps: (al'=al+1) & (oil'=oil-2) +
35 (1-Ps): (s'=2) & (oil'=oil-1);
36
37 // decision in Area B
38 [tick2] (s = 2) & (t = 0) & (bl < Nb) & (oil > 0)
39 & (w2 = 0) -> (bl'=bl+1) & (oil'=oil-1);
40 [tick2] (s = 2) & (t = 0) & (bl = Nb) & (al < Na)
41 & (oil > 0) -> (s'=1) & (oil'=oil-1);
42 [tick2] (s = 2) & (t = 0) & (bl < Nb) & (al < Na)
43 & (oil > 1) & (w2 = 1) & (w1 = 1)
44 -> (bl'=bl+1) & (oil'=oil-2);
45 [tick2] (s = 2) & (t = 0) & (bl < Nb) & (al < Na)
46 & (oil > 1) & (w2 = 1) & (w1 = 0)
47 -> Ps: (bl'=bl+1) & (oil'=oil-2) + (1-Ps): (s'=2) & (oil'=oil-1);
48 [tick2] (s > 0) & (s < 3) & (oil = 0) -> (s'=3);
49
50 endmodule
51
52 module environment
53   t: [0..1] init 1; // control weather change
54
55 [tick1] (s > 0) & (s < 3) & (t = 1) -> (t' = 0);
56 [tick2] (s > 0) & (s < 3) & (t = 0) -> (t' = 1);
57
58 endmodule
59
60 module weather1
61   w1 : [0..1];
62
63 [tick1] (s > 0) & (s < 3) & (t = 1) -> Pa: (w1' = 1) + (1-Pa): (w1' = 0);
64
65 endmodule
66
67 module weather2
68   w2 : [0..1];
69
70 [tick1] (s > 0) & (s < 3) & (t = 1) -> Pb: (w2' = 1) + (1-Pb): (w2' = 0);
71
72 endmodule

```

REFERENCES

- [1] K. J. Åström, *Autonomous Control*. Berlin: Springer-Verlag, 1992, vol. Future Tendencies in Computer Science, Edts: A. Bensoussan and J.P. Verius, Vol. 653, Lecture Notes in Computer Science.
- [2] K. J. Åström and R. M. Murray, *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2010.
- [3] K. W. Ng and C.-K. Luk, “I+: A multiparadigm language for object-oriented declarative programming,” *Computer Languages*, vol. 21, no. 2, pp. 81–100, 1995.
- [4] P. Ridao, J. Batlle, and M. Carreras, “O 2 ca 2, a new object oriented control architecture for autonomy: the reactive layer,” *Control Engineering Practice*, vol. 10, no. 8, pp. 857–873, 2002.
- [5] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe, “Modeling and designing heterogeneous systems,” in *Concurrency and Hardware Design*. Springer, 2002, pp. 228–273.
- [6] A. Pinto, A. L. Sangiovanni-Vincentelli, L. P. Carloni, and R. Passerone, “Interchange formats for hybrid systems: Review and proposal,” in *Hybrid Systems: Computation and Control*. Springer, 2005, pp. 526–541.
- [7] B. I. Silva, O. Stursberg, B. H. Krogh, and S. Engell, “An assessment of the current status of algorithmic approaches to the verification of hybrid systems,” in *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, vol. 3. IEEE, 2001, pp. 2867–2874.
- [8] S. M. Veres, L. Molnar, N. K. Lincoln, and C. Morice, “Autonomous vehicle control systems - a review of decision making,” vol. 225, no. 2, pp. 155–195, 2011.
- [9] M. Wooldridge, *An Introduction to MultiAgent Systems*. Chichester: Wiley, 2002.
- [10] M. Wooldridge and N. R. Jennings, “Intelligent agents: theory and practice,” *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1995.
- [11] E. Gat, “On three-layer architectures,” *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pp. 195–210, 1998, mIT Press.
- [12] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, “An architecture for autonomy,” *The International Journal of Robotics Research*, vol. 17, no. 4, pp. 315–337, 1998.
- [13] R. Simmons and D. Apfelbaum, “A task description language for robot control,” in *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, vol. 3. IEEE, 1998, pp. 1931–1937.
- [14] I. A. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I.-H. Shu *et al.*, “Claraty: Challenges and steps toward reusable robotic software,” *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 23–30, 2006.
- [15] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, “The claraty architecture for robotic autonomy,” in *Aerospace Conference, 2001, IEEE Proceedings.*, vol. 1. IEEE, 2001, pp. 1–121.
- [16] R. A. Brooks, “Elephants don’t play chess,” *Robotics and autonomous systems*, vol. 6, no. 1, pp. 3–15, 1990.
- [17] [Online]. Available: <http://www.moos-ivp.org>
- [18] M. R. Benjamin, H. Schmidt, M. P. Newman, and J. J. Leonard, *Unmanned Marine Vehicle Autonomy with MOOS-IvP*. Springer, 2012.
- [19] M. P. Newman, “Moos - a mission oriented operating suite,” Massachusetts Institute of Technology, Tech. Rep. OE2003-07, 2003.
- [20] M. R. Benjamin, “The interval programming model for multi-objective decision making,” Massachusetts Institute of Technology, Tech. Rep. AIM-2004-021, 2004.
- [21] R. H. Bordini, J. F. Hubner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. Chichester: Wiley, 2007.
- [22] M. P. Georgeff and A. L. Lansky, “Procedural knowledge,” *Proceedings of the IEEE*, vol. 74, no. 10, pp. 1383–1398, 1986.
- [23] —, “Reactive reasoning and planning,” in *AAAI*, vol. 87, 1987, pp. 677–682.
- [24] A. S. Rao, “Agentspeak (I): Bdi agents speak out in a logical computable language,” in *Agents Breaking Away*. Springer, 1996, pp. 42–55.
- [25] R. H. Bordini and J. F. Hubner, *Jason, A Java-based interpreter for an extended version of AgentSpeak*, 2007, manual version 0.9.5.
- [26] F. L. Bellifemine, A. Poggi, G. Rimassa, and P. Turci, “An object-oriented framework to realize agent systems,” in *WOA*, 2000, pp. 52–57.
- [27] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing multi-agent systems with JADE*. John Wiley & Sons, 2007, vol. 7.
- [28] Y. Shoham, “Agent-oriented programming,” *Artificial intelligence*, vol. 60, no. 1, pp. 51–92, 1993.
- [29] N. K. Lincoln and S. M. Veres, “Natural language programming of complex robotic bdi agents,” *Intelligent and Robotic Systems*, vol. 71, no. 2, pp. 211–230, 2013.
- [30] M. Kwiatkowska, G. Norman, and D. Parker, “Prism 4.0: Verification of probabilistic real-time systems,” in *Computer aided verification*. Springer, 2011, pp. 585–591.
- [31] [Online]. Available: <http://www.prismmodelchecker.org/>
- [32] T. Han, J. Katoen, and B. Damman, “Counterexample generation in probabilistic model checking,” *IEEE Trans. Software Eng.*, vol. 35, no. 2, pp. 241–257, 2009.